Insight:

Blocks, Closures, Bindings

In this lecture, you will learn some insights about blocks in Ruby.
You will also demystify one of the common tricks (&:symbol) used in Ruby on Rails.

Some block examples

```ruby
lambda { |x| "some code" }                          # Using 'lambda'

->(x) { "shorthand version of lambda block" }       # Shorthand syntax for lambda

proc do                                             # Similar to 'lambda', but creates 'proc'
    puts "Multiline block"                          # Using do..end syntax
    puts "with do...end"
end

blk = Proc.new { "some block content" }             # Creating an instnace of Proc class
blk.call                                            # Returns: "some block content"

def foo(&block)                                     # Method that takes block
    yield                                           # 'yield' executes block
    block.call if block_given?                      # block_given? checks if block was given
end
```

Lambda & Procs are not blocks.

They are their object representation (Proc class).

Some block examples

```ruby
do
    "this is block"
end

lambda { "takes block and returns Proc class instance" }    # #<Proc:0x007fedeb8fb390>

a = do
    "you can't assign block to a variable"
end                                                         # SyntaxError: (irb):2: syntax error, unexpected keyword_do_block

def foo
    yield
end

foo { "but you can pass it to a method" }                   # "but you pass it to a method"

def bar(&my_block)
    my_block.call
end

bar do
    "how it is that we can pass it as an argument to
    a block if we can't assign it to a variable?"
end
```

Converting block into Proc & back again

```ruby
def foo(&block)
    puts "Block was given" if block_given?
end

def bar(&my_block)
    puts my_block.class                    # Proc
    foo(&my_block)                         # 'my_block' is converted from Proc instance back into block
end

bar { "some block" }                       # Proc
                                           # Block was given
```

“&” calls “.to_proc” method

```ruby
class Foo
    def to_proc
        Proc.new { "This is Foo#to_proc." }
    end
end

def bar(&my_block)
    my_block.call
end

bar(&Foo.new)                                   # "This is Foo#to_proc."

["one", "two"].map(&:upcase)                    # ["ONE", "TWO"]


class Symbol
    def to_proc
        Proc.new { |x| x.send(self) }           # we will talk about 'send' in the next Section
    end
end
```

Blocks are evaluated in the scope where they were defined.

Not where they are run.

Closure

```ruby
foo = "foo in top level."

my_block = lambda do |x|
    puts x
    puts foo
end

class MyClass
    def initialize(&block)
       @block = block
       @instance_var = "MyClass instance variable."
    end

    def bar
       # There is no access to top level "foo" variable from here.
       # If you try to access it here, you would get an error:
       # NameError: undefined local variable or method `foo' for #<MyClass:0x007fbb04a752f8>
       @block.call(@instance_var)
    end
end

# 'my_block' is holding an instance of Proc class. We need to convert it into block,
# because the method is expecting a block and has no parameters to receive it as an
# argument to the method.
baz = MyClass.new(&my_block)
baz.bar                                                 # MyClass instance variable.
                                                        # Foo in top level.

foo = "Changed value of foo."
baz.bar                                                 # MyClass instance variable.
                                                        # Changed value of foo.
```